THE INTERCAL PROGRAMMING LANGUAGE REFERENCE MANUAL

Donald R. Woods
and
James M. Lyon

1. INTRODUCTION

The names you are about to ignore are true. However, the story has been changed significantly. Any resemblance of the programming language portrayed here to other programming languages, living or dead, is purely coincidental.

1.1. Origin and Purpose

The INTERCAL programming language was designed the morning of May 26, 1972 by Donald R. Woods and James M. Lyon, at Princeton University. Exactly when in the morning will become apparent in the course of this manual. It was inspired by one ambition; to have a compiler language which has nothing at all in common with any other major language. By 'major' was meant anything with which the authors were at all familiar, e.g., FORTRAN, BASIC, COBOL, ALGOL, SNOBOL, SPITBOL, FOCAL, SOLVE, TEACH, APL, LISP, and PL/I. For the most part, INTERCAL has remained true to this goal, sharing only the basic elements such as variables, arrays, and the ability to do I/O, and eschewing all conventional operations other than the assignment statement (FORTRAN "=").

1.2. Acronym

The full name of the compiler is "Compiler Language With No Pronounceable Acronym", which is, for obvious reasons, abbreviated "INTERCAL".

1.3. Acknowledgments

The authors are deeply indebted to Eric M. Van and Daniel J. Warmenhoven, without whose unwitting assistance this manual would still have been possible.

2. FUNDAMENTAL CONCEPTS

In this section an attempt is made to describe how and why INTERCAL may be used; i.e., what it is like and what it is good for.

2.1. Sample Program

Shown below is a relatively simple INTERCAL program which will read in 32-bit unsigned integers, treat them as signed, 2's-complement numbers, and print out their absolute values. The program exits if the absolute value is zero. Note in particular the inversion routine (statements 6 through 14), which could be greatly simplified if the subroutine library (see section 5) were used.

A more detailed analysis of a program is made in section 6 of this manual.

```
DO (5) NEXT
(5) DO FORGET #1
PLEASE WRITE IN :1
DO .1 <- '∀":1~'#32768¢#0'"¢#1'~#3
DO (1) NEXT
DO :1 <- "'∀":1~'#65535¢#0'"¢#65535'
     ~'#0¢#65535'"¢"'∀":1~'#0¢#65535'"
     ¢#65535'~'#0¢#65535'"
DO :2 <- #1
PLEASE DO (4) NEXT
(4) DO FORGET #1
DO .1 <- "∀':1~:2'¢#1"~#3
DO :1 <- "'∀":1~'#65535¢#0'"¢":2~'#65535
     ¢#0'"'~'#0¢#65535'"¢"'∀":1~'#0
     ¢#65535'"¢":2~'#65535¢#0'"'~'#0¢#65535'"
DO (1) NEXT
DO :2 <- ":2~'#0¢#65535'"
     ¢"'":2~'#65535¢#0'"¢#0'~'#32767¢#1'"
DO (4) NEXT
```

```
(2) DO RESUME .1
(1) PLEASE DO (2) NEXT
PLEASE FORGET #1
DO READ OUT :1
PLEASE DO .1 <- '∀"':1~:1'~#1"¢#1'~#3
DO (3) NEXT
PLEASE DO (5) NEXT
(3) DO (2) NEXT
PLEASE GIVE UP
```

2.2. Uses for INTERCAL

INTERCAL's main advantage over other programming languages is its strict simplicity. It has few capabilities, and thus there are few restrictions to be kept in mind. Since it is an exceedingly easy language to learn, one might expect it would be a good language for initiating novice programmers. Perhaps surprising, than, is the fact that it would be more likely to initiate a novice into a search for another line of work. As it turns out, INTERCAL is more useful (which isn't saying much) as a challenge to professional programmers. Those who doubt this need only refer back to the sample program in section 2.1. This 22-statement program took somewhere from 15 to 30 minutes to write, whereas the same objectives can be achieved by single-statement programs in either SNOBOL;

```
PLEASE INPUT POS(0) ('-' ! '')
+ (SPAN('0123456789') $ OUTPUT)
+ *NE(OUTPUT) :S(PLEASE)F(END)
```

or APL;

```
[1] →0≠□←⍞□
```

Admittedly, neither of these is likely to appear more intelligible to anyone unfamiliar with the languages involved, but they took roughly 60 seconds and 15 seconds, respectively, to write. Such is the overwhelming power of INTERCAL!

The other major importance of INTERCAL lies in its seemingly inexhaustible capacity for amazing one's fellow programmers, confounding programming shop managers, winning friends, and influencing people. It is a well-known and oft-demonstrated fact that a person whose work is incomprehensible is held in high esteem. For example, if one were to state that the simplest way to store a value of 65536 in a 32-bit INTERCAL variable is:

```
DO :1 <- #0¢#256
```

any sensible programmer would say that that was absurd. Since this is indeed the simplest method, the programmer would be made to look foolish in front of his boss, who would of course happened to turn up, as bosses are wont to do. The effect would be no less devastating for the programmer having been correct.

3. DESCRIPTION

The examples of INTERCAL programming which have appeared in the preceding sections of this manual have probably seemed highly esoteric to the reader unfamiliar with the language. With the aim of making them more so, we present here a description of INTERCAL.

3.1. Variables

INTERCAL allows only 2 different types of variables, the **16-bit integer** and the **32-bit integer.** These are represented by a spot (.) or two-spot (:), respectively, followed by any number between 1 and 65535, inclusive. These variables may contain only non-negative numbers; thus they have the respective ranges of values: 0 to 65535 and 0 to 4294967295. Note: .123 and :123 are two distinct variables. On the other hand, .1 and .0001 are identical. Furthermore, the latter may NOT be written as 1E-3.

3.2. Constants

Constants are 16-bit values only and may range from 0 to 65535. They are prefixed by a mesh (#). Caution! Under no circumstances confuse the mesh with the interleave operator, except under confusing circumstances!

3.3. Arrays

Arrays are represented by a tail (,) for 16-bit values, or a hybrid (;) for 32-bit values, followed by a number between 1 and 65535, inclusive. The number is suffixed by the word SUB, followed by the subscripts, separated optionally by spaces. Subscripts may be any expressions, including those involving subscripted variables. This occasionally leads to ambiguous constructions, which are resolved as discussed in section 3.4.3. Definition of array dimensions will be discussed later in greater detail, since discussing it in less detail would be difficult. As before, ,123 and ;123 are distinct. In summary, .123, :123, #123, ,123, and :123 are all distinct.

3.4. Operators

INTERCAL recognizes 5 operators--2 binary and 3 unary. Please be kind to our operators: they may not be very intelligent, but they're all we've got. In a sense, all 5 operators are binary, as they are all bit-oriented, but it is not our purpose here to quibble about bits of trivia.

3.4.1. Binary Operators

The binary operators are **interleave** (also called **mingle**) and **select**, which are represented by a change (¢) and a sqiggle [sic] (~), respectively.

The interleave operator takes two 16-bit values and produces a 32-bit result by alternating the bits of the operands. Thus, #65535¢#0 has the 32-bit binary form 101010....10 or 2863311530 decimal, while #0¢#65535 = 0101....01 binary = 1431655765 decimal, and #255¢#255 is equivalent to #65535.

The select operator takes from the first operand whichever bits correspond to 1's in the second operand, and packs these bits to the right in the result. Both operands are automatically padded on the left with zeros to 32 bits before the selection takes place, so the variable types are unrestricted. If more than 16 bits are selected, the result is a 32-bit value, otherwise it is a 16-bit value. For example, #179~#201 (binary value 10110011~11001001) selects from the first argument the 8th, 7th, 4th, and 1st from last bits, namely, 1001, which = 9. But #201~#179 selects from binary 11001001 the 8th, 6th, 5th, 2nd, and 1st from last bits, giving 10001 = 17. #179~#179 has the value 31, while #201~#201 has the value 15.

Perhaps a simpler way of understanding the operation of the select operator would be to examine the logic diagram on the following page (Figure 1), which performs the select operation upon two 8-bit values, A and B. The gates used are Warmenhovian logic gates, which means the outputs have four possible values: low, high, undefined (value of an uninitialized flip-flop), and oscillating (output of a NOR gate with one input low and the other input connected to the output). These values are represented symbolically by '0', '1', '?', and '∅'. Note in particular that, while NOT-0 is 1 and NOT-1 is 0 as in two-valued logic, NOT-? is ? and NOT-∅ is ∅. The functions of the various gates are listed in Table 1.

Figure 1 goes here with above page number.

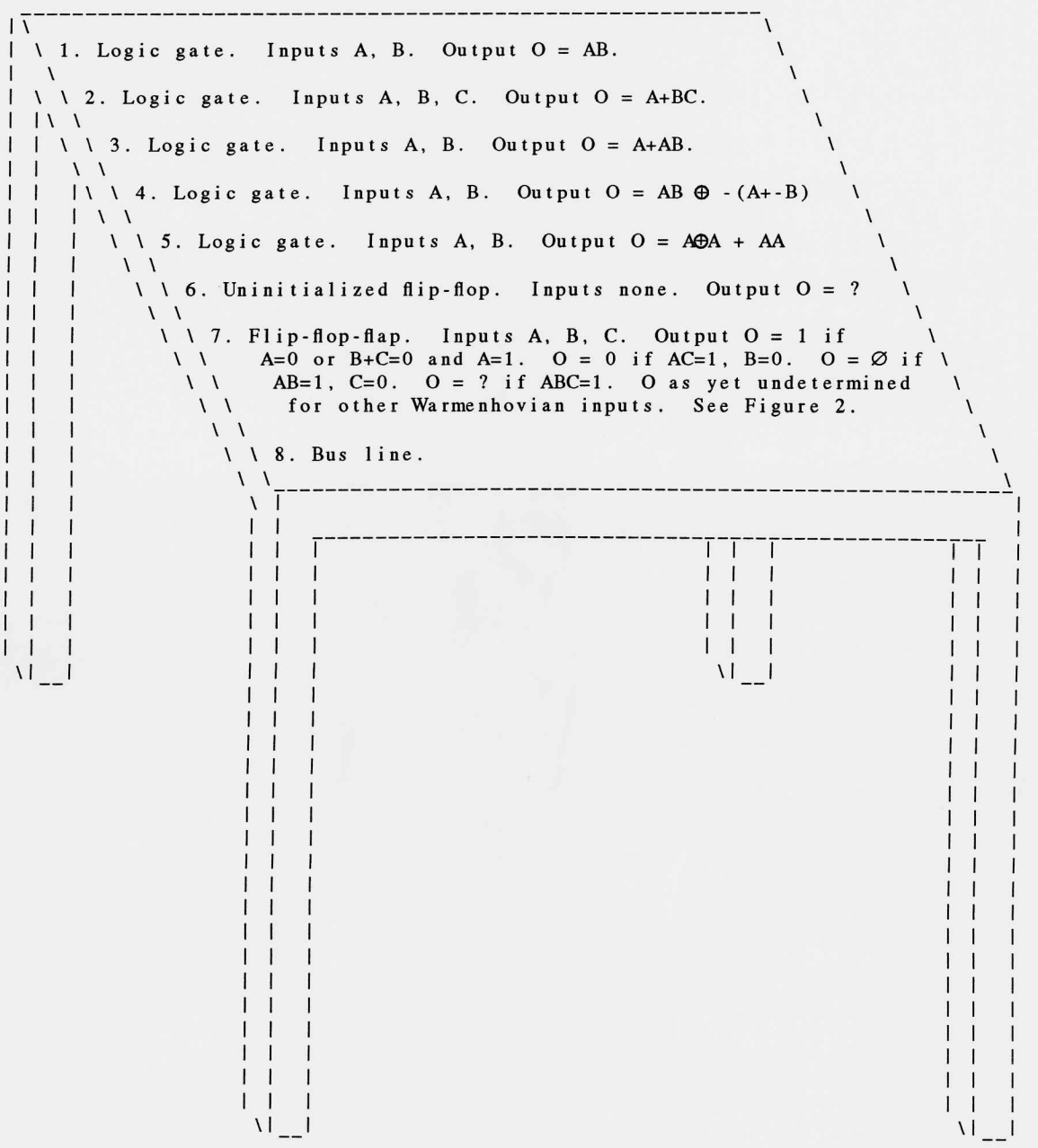

Table 1. Logical (and other) functions.

Figure 2 goes here with above page number.

Rest of Figure 1 goes here with above page number.

3.4.2. Unary Operators

The unary operators are & (logical **AND**), V (logical **OR**), and ∀ (logical **XOR**). This last character is obtained by overpunching a worm (–) on a V (V). The operator is inserted between the spot, two-spot, mesh, or what-have-you, and the integer, thus: .&123, #∀123. Multiple unary operators may not be concatenated, thus the form #∀&123 is invalid. This will be covered later when precedence is discussed. These operators perform their respective logical operations on all pairs of adjacent bits, the result from the first and last bits going into the first bit of the result. The effect is that of rotating the operand one place to the right and ANDing, ORing, or XORing with its initial value. Thus, #&77 (binary = 1001101) is binary 0000000000000100 = 4, #V77 is binary 1000000001101111 = 32879, and #∀77 is binary 1000000001101011 = 32875.

3.4.3. Precedence

Precedence of operators is as follows:

(The remainder of this page intentionally left blank)[1]

[1] Keep in mind that the aim in designing INTERCAL was to have no precedents.

This precedence (or lack thereof) may be overruled by grouping expressions between pairs of sparks (') or rabbit-ears ("). Thus '#165¢#203'~#358 (binary value '10100101¢11001011'~101100110) has the value 15, but #165¢'#203~#358' has the value 34815, and #165¢#203~#358 is invalid syntax and is completely valueless (except perhaps as an educational tool to the programmer). A unary operator is applied to a sparked or rabbit-eared expression by inserting the operator immediately following the opening spark or ears. Thus, the invalid expression #∀&123, which was described earlier, could be coded as '∀#&123' or '∀"&#123"'. Note: In the interests of simplifying the sometimes overly-complex form of expressions, INTERCAL allows a spark-spot combination ('.) to be replaced with a wow (!). Thus '.1~.2' is equivalent to !1~.2', and 'V.1¢.2' is equivalent to "V!1¢.2'".

Combining a rabbit-ears with a spot to form a rabbit (:) is not permitted, although the programmer is free to use it should he find an EBCDIC reader which will properly translate a 12-3-7-8 punch.

Sparks and/or rabbit-ears must also be used to distinguish among such otherwise ambiguous subscripted and multiply-subscripted expressions as:

```
,1 SUB #1 ~ #2
,1 SUB ,2 SUB #1 #2 #3
,1 SUB " ,2 SUB " ,3 SUB #1 " #2 " " #3 "
```

The third case may be isolated into either of its possible interpretations by simply changing some pairs of rabbit-ears to sparks, instead of adding more ears (which would only confuse the issue further). Ambiguous cases are defined as those for which the compiler being used finds a legitimate interpretation which is different from that which the user had in mind. See also section 8.1.

4. STATEMENTS

In this section is described the format of INTERCAL statements.

4.1. General Format

Statements may be entered in 'free format'. That is, more than one statement may occur on a single card, and a statement may begin on one card and end on a later one. Note that if this is done, all intervening cards and portions thereof must be part of the same statement. That this restriction is necessary is immediately apparent from the following example of what might occur if statements could be interlaced.

```
DO .1 <- ".1¢'&:51~"#∀1c!12~;&75SUB"∀'V.1~
DO .2 <- '"!1¢"&';∀79SUB",&7SUB:173"'~!∀9¢
.2'¢,&1SUB:5~#33578"'"'"~'#65535¢"∀'V#&85'"'
#8196'"'~.1"¢.2'~'#&5¢"'#1279¢#4351'~#65535"'
```

The above statements are obviously meaningless. (For that matter, so are the statements

```
DO .1 <- ".1¢"&:51~"#∀1¢!12~;&75SUB"∀'V.1~
.2'¢,&1SUB:5~#333578"'"'"~#65535¢"∀'V#&85'"'
DO .2 <- '"!1¢"&';∀79SUB",&7SUB:173"'~!∀9¢
#8196'"'~.1"¢.2'~'#&5¢"'#1279¢#4351'~#65535"'
```

but this is not of interest here.)

Spaces may be used freely to enhance program legibility (or at least reduce program illegibility), with the restriction that no word of a statement identifier (see section 4.3) may contain any spaces.

4.2. Labels

A statement may begin with a **logical line label** enclosed in wax-wane pairs (()). A statement may not have more than one label, although it is possible to omit the label entirely. A line label is any integer from 1 to 65535, which must be unique within each program. The user is cautioned, however, that many line labels between 1000 and 1999 are used in the INTERCAL System Library functions.

4.3. Identifiers and Qualifiers

After the line label (if any), must follow one of the following statement identifiers: DO, PLEASE, or PLEASE DO. These may be used interchangeably to improve the aesthetics of the program. The identifier is then followed by either, neither, or both of the following optional parameters (qualifiers): (1) either of the character strings NOT or N'T, which causes the statement to be automatically abstained from (see section 4.4.9) when execution begins, and (2) a number between 0 and 100, preceded by a double-oh-seven (%), which causes the statement to have only the specified percent chance of being executed each time it is encountered in the course of execution.

4.4. Statements

Following the qualifiers (or, if none are used, the identifier) must occur one of the 13 valid operations. (Exception: see section 4.5.) These are described individually in sections 4.4.1 through 4.4.13.

4.4.1. Calculate

The INTERCAL equivalent of the half-mesh (=) in FORTRAN, BASIC, PL/I, and others, is represented by an angle (<) followed by a worm (-). This combination is read 'gets'. 32-bit variables may be assigned 16-bit values, which are padded on the left with 16 zero bits. 16-bit variables may be assigned 32-bit values only if the value is less than 65535. Thus, to invert the least significant bit of the first element of 16-bit 2-dimensional array number 1, one could write:

```
,1SUB#1#1 <- '¥,1SUB#1#1¢#1'~'#0¢#65535'
```

Similarly to SNOBOL and SPITBOL, INTERCAL uses the angle-worm to define the dimensions of arrays. An example will probably best describe the format. To define 32-bit array number 7 as 3-dimensional, the first dimension being seven, the second being the current value of 16-bit variable number seven, and the third being the current value of the seventh element of 16-bit array number seven (which is one-dimensional) mingled with the last three bits of 32-bit variable number seven, one would write (just before they came to take him away):

```
;7 <- #7 BY .7 BY ",7SUB#7"¢':7~#7'
```

This is, of course, different from the statement:

```
;7 <- #7 BY .7 BY ,7SUB"#7¢':7~#7'"
```

INTERCAL also permits the redefining of array dimensioning, which is done the same way as is the initial dimensioning. All values of items in an array are lost upon redimensioning, unless they have been STASHed (see section 4.4.5), in which case restoring them also restores the old dimensions.

4.4.2. NEXT

The NEXT statement is used both for subroutine calls and for unconditional transfers. This statement takes the form:

```
DO (label) NEXT
```

(or, of course,

```
PLEASE DO (label) NEXT
```

etc.), where (label) represents any logical line label which appears in the program. The effect of such a statement is to transfer control to the statement specified, and to store in a push down list (which is initially empty) the location from which the transfer takes place. Items may be removed from this list and may be discarded or used to return to the statement immediately following the NEXT statement. These operations are described in sections 4.4.3 and 4.4.4 respectively. The programmer is generally advised to discard any stack entries which he does not intend to utilize, since the stack has a maximum depth of 79 entries. A program's attempting to initiate an 80th level of NEXTing will result on the fatal error message, "PROGRAM HAS DISAPPEARED INTO THE BLACK LAGOON."

4.4.3. FORGET

The statement PLEASE FORGET exp, where exp represents any expression (except colloquial and facial expressions), causes the expression to be evaluated, and the specified number of entries to be removed from the NEXTing stack and discarded. An attempt to FORGET more levels of NEXTing than are currently stacked will cause the stack to be emptied, and no error condition is indicated. This is because the condition is not considered to be an error. As described in section 4.4.2, it is good programming practice to execute a DO FORGET #1 after using a NEXT statement as an unconditional transfer, so that the stack does not get cluttered up with unused entries:

```
    DO (123) NEXT
    .
    .
(123) DO FORGET #1
```

4.4.4. RESUME

The statement PLEASE RESUME exp has the same effect as FORGET, except that program control is returned to the statement immediately following the NEXT statement which stored in the stack the last entry to be removed. Note that a rough equivalent of the FORTRAN computed GO TO and BASIC ON exp GO TO is performed by a sequence of the form:

```
    DO (1) NEXT
    .
    .
    (1) DO (2) NEXT
    PLEASE FORGET #1
    .
    .
    (2) DO RESUME .1
```

Unlike the FORGET statement, an attempt to RESUME more levels of NEXTing than been stacked will cause program termination. See also section 4.4.11.

4.4.5. STASH

Since subroutines are not explicitly implemented in INTERCAL, the NEXT and RESUME statements must be used to execute common routines. However, as these routines might use the same variables as the main program, it is necessary for them to save the values of any variables whose values they alter, and later restore them. This process is simplified by the STASH state ment, which has the form DO STASH list, where list represents a string of one or more variable or array names, separated by intersections (+). Thus

```
PLEASE STASH .123+:123+,123
```

stashes the values of two variables and one entire array. The values are left intact, and copies thereof are saved for later retrieval by (what else?) the RETRIEVE statement (see section 4.4.6). It is not possible to STASH single array items.

4.4.6. RETRIEVE

PLEASE RETRIEVE list restores the previously STASHed values of the variables and arrays named in the list. If a value has been stashed more than once, the most recently STASHed values are RETRIEVEd, and a second RETRIEVE will restore the second most recent values STASHed. Attempting to RETRIEVE a value which has not been STASHed will result in the error message, "THROW STICK BEFORE RETRIEVING."

4.4.7. IGNORE

The statement DO IGNORE list causes all subsequent statements to have no effect upon variables and/or arrays named in the list. Thus, for example, after the sequence

```
DO .1 <- #1
PLEASE IGNORE .1
DO .1 <- #0
```

16-bit variable number 1 would have the value 1, not 0. Inputting (see section 4.4.12) into an IGNOREd variable also has no effect. The condition is annulled via the REMEMBER statement (see section 4.4.8). Note that, when a variable is being IGNOREd, its value, though immutable, is still available for use in expressions and the like.

4.4.8. REMEMBER

PLEASE REMEMBER list terminates the effect of the IGNORE statement for all variables and/or arrays named in the list. It does not matter if a variable has been IGNOREd more than once, nor is it an error if the variable has not been IGNOREd at all.

4.4.9. ABSTAIN

INTERCAL contains no simple equivalent to an IF statement or computed GO TO, making it difficult to combine similar sections of code into a single routine which occasionally skips around certain statements. The IGNORE statement (see section 4.4.7) is helpful in some cases, but a more viable method is often required. In keeping with the goal of INTERCAL having nothing in common with any other language, this is made possible via the ABSTAIN statement.

This statement takes on one of two forms. It may not take on both at any one time. DO ABSTAIN FROM (label) causes the statement whose logical line label is (label) to be abstained form. PLEASE ABSTAIN FROM gerund list causes all statements of the specified type(s) to be abstained from, as in

```
PLEASE ABSTAIN FROM STASHING
PLEASE ABSTAIN FROM IGNORING + FORGETTING
PLEASE ABSTAIN FROM NEXTING
or PLEASE ABSTAIN FROM CALCULATING
```

Statements may also be automatically abstained from at the start of execution via the NOT or N'T parameter (see section 4.3).

If, in the course of execution, a statement is encountered which is being abstained from, it is ignored and control passes to the next statement in the program (unless it, too, is being abstained from).

The statement DO ABSTAIN FROM ABSTAINING is perfectly valid, as is DO ABSTAIN FROM REINSTATING (although this latter is not usually recommended). However, the statement DO ABSTAIN FROM GIVING UP is not accepted, even though DON'T GIVE UP is.

4.4.10. REINSTATE

The REINSTATE statement, like the ABSTAIN, takes as an argument either a line label or a gerund list. No other form of argument is permitted. For example, the following is an invalid argument:

```
Given: x≠0, y≠0,  Prove: x+y=0
Since x≠0, then x+1≠1, x+a≠a, x+y≠y.
But what is y? y is anything but 0.
Thus x+y ≠ anything but 0.
Since x+y cannot equal anything but 0, x+y=0.

                    Q.E.D.
```

REINSTATEment nullifies the effects of an abstention. Either form of REINSTATEment can be used to "free" a statement, regardless of whether the statement was abstained from by gerund list, line label, or NOT. Thus, PLEASE REINSTATE REINSTATING is not necessarily an irrelevant statement, since it might free a DON'T REINSTATE command or a REINSTATE the line label of which was abstained from. However, DO REINSTATE GIVING UP is invalid, and attempting to REINSTATE a GIVE UP statement by line label will have no effect. Note that this insures that DON'T GIVE UP will always be a "do-nothing" statement.

4.4.11. GIVE UP

PLEASE GIVE UP is used to exit from a program. It has the effect of a PLEASE RESUME #80. DON'T GIVE UP, as noted in section 4.4.10, is effectively a null statement.

4.4.12. Input

Input is accomplished with the statement DO WRITE IN list, where list represents a string of variables and/or elements or arrays, separated by intersections. Numbers are represented on cards, each number on a separate card, by spelling out each digit (in English) and separating the digits with one or more spaces. A zero (0) may be spelled as either ZERO or OH. Thus the range of (32-bit) input values permissible extends from ZERO (or OH) through FOUR TWO NINE FOUR NINE SIX SEVEN TWO NINE FIVE.

Attempting to write in a value greater than or equal to SIX FIVE FIVE three six for a 16-bit variable will result in the error message, "DON'T BYTE OFF MORE THAN YOU CAN CHEW."

4.4.13. Output

Values may be output to the printer, one value per line, via the statement DO READ OUT list, where the list contains variables, array elements, and/or constants. Output is in the form of "extended" Roman numerals (also called "butchered" Roman numerals), with an overline (‾) indicating the value below is "times 1000", and lower-case letters indicating "times 1000000". Zero is indicated by an overline with no character underneath. Thus, the range of (32-bit) output values possible is from ‾ through $\overline{\text{i}\overline{\text{v}}}$ccxcivC$\overline{\text{M}}\overline{\text{L}}\overline{\text{X}}\overline{\text{V}}\overline{\text{I}}$CCXCV. Note: For values whose residues modulo 1000000 are less than 4000, M is used to represent 1000; for values whose residues are 4000 or greater, $\overline{\text{I}}$ is used. Thus #3999 would read out as MMMIM while #4000 would read out as $\overline{\text{I}}\overline{\text{V}}$. Similar rules apply to the use of $\overline{\text{M}}$ and i for 1000000, and to that of m and $\overline{\text{i}}$ for 1000000000.

4.5. Comments

Unrecognizable statements, as noted in section 7, are flagged with a splat (*) during compilation, and are not considered fatal errors unless they are encountered during execution, at which time the statement (as input at compilation time) is printed and execution is terminated. This allows for an interesting (and, by necessity, unique) means of including comments in an INTERCAL listing. For example, the statement:

```
*     PLEASE NOTE THAT THIS LINE HAS NO EFFECT
```

will be ignored during execution due to the inclusion of the NOT qualifier. User-supplied error messages are also easy to implement:

```
*     DO SOMETHING ABOUT OVERFLOW IN ;3
```

as are certain simple conditional errors:

```
* (123) DON'T YOU REALIZE THIS STATEMENT SHOULD ONLY BE ENCOUNTERED
        ONCE?
  PLEASE REINSTATE (123)
```

This pair of statements will cause an error exit the second time they are encountered. Caution!! The appearance of a statement identifier in an intended comment will be taken as the beginning of a new statement. Thus, the first example on the preceding page could not have been:

```
*     PLEASE NOTE THAT THIS LINE DOES NOTHING
```

The third example, however, is valid, despite the appearance of two cases of D-space-O, since INTERCAL does not ignore extraneous spaces in statement identifiers.

5. SUBROUTINE LIBRARY

INTERCAL provides several built-in subroutines to which control can be transferred to perform various operations. These operations include many useful functions which are not easily representable in INTERCAL, such as addition, subtraction, etc.

5.1. Usage

In general, the operands are .1, .2, etc., or :1, :2, etc., and the result(s) are stored in what would have been the next operand(s). For instance, one routine adds .1 to .2 and store the sum in .3, with .4 being used to indicate overflow. All variables not used for results are left unchanged.

5.2. Available Functions

At the time of this writing, only the most fundamental operations are offered in the library, as a more complete selection would require prohibitive time and coree to implement. These functions, along with their corresponding entry points (entered via DO (entry) NEXT) are listed below.

```
(1000) .3 <- .1 plus .2, error exit on overflow
(1009) .3 <- .1 plus .2
   .4 <- #1 if no overflow, else .4 <- #2
(1010) .3 <- .1 minus .2, no action on overflow
(1020) .1 <- .1 plus #1, no action on overflow
(1030) .3 <- .1 times .2, error exit on overflow
(1039) .3 <- .1 times .2
   .4 <- #1 if no overflow, else .4 <- #2
(1040) .3 <- .1 divided by .2
   .3 <- #0 if .2 is #0
(1050) .2 <- :1 divided by .1, error exit on overflow
   .2 <- #0 if .1 is #0

(1500) :3 <- :1 plus :2, error exit on overflow
(1509) :3 <- :1 plus :2
   :4 <- #1 if no overflow, else :4 <- #2
(1510) :3 <- :1 minus :2, no action on overflow
(1520) :1 <- .1 concatenated with .2
(1525) This subroutine is intended solely for internal
   use within the subroutine library and is therefore
   not described here. Its effect is to shift .3
   logically 8 bits to the left.
(1530) :1 <- .1 times .2
(1540) :3 <- :1 times :2, error exit on overflow
(1549) :3 <- :1 times :2
   :4 <- #1 if no overflow, else :4 <- #2
(1550) :3 <- :1 divided by :2
   :3 <- #0 if :2 is #0

(1900) .1 <- uniform random no. from #0 to #65535
(1910) .2 <- normal random no. from #0 to .1, with
         standard deviation .1 divided by #12
```

6. PROGRAMMING HINTS

For the user looking to become more familiar with the INTERCAL language, we present in this section an analysis of a complex program, as well as some suggested projects for the ambitious programmer.

Considering the effort involved in writing an INTERCAL program, it was decided in putting together this manual to use an already existing program for instructive analysis. Since there was only one such program available, we have proceeded to use it. It is known as the "INTERCAL System Library."

6.1. Description

The program listing begins on the second page following. It is in the same format as would be produced by the Princeton INTERCAL compiler in FORMAT mode with WIDTH=62 (see section 8). For a description of the functions performed by the Library, see section 5.2.

6.2. Analysis

We shall not attempt to discuss here the algorithms used, but rather we shall point out some of the general techniques applicable to a wide range of problems.

Statements 10, 14, 15, and 26 make up a virtual "computed GO TO". When statement 10 is executed, control passes eventually to statement 16 or 11, depending on whether .5 contains #1 or #2, respectively. The value of .5 is determined in statement 9, which demonstrates another handy technique. To turn an expression, exp, with value #0 or #1, into #1 or #2 (for use in a "GO TO"), use "∀'exp'¢#1"~#3. To reverse the condition (i.e., convert #0 to #2 and leave #1 alone) use "∀'exp'¢#2"~#3.

Certain conditions are easily checked. For example, to test for zero, select the value from itself and select the bottom bit (see statement 54). To test for all bits being 1's, select the value from itself and select the top bit (see statement 261). The test for greater than, performed in statements 192 and 193 on 32-bit values, employs binary logical operations, which are performed as follows:

for 16-bit values or, for 32-bit values:

```
"'∀":1~'#65535¢30'"¢":2~'#65535¢#0'"'~'#0
¢#65535'"¢"'∀":1~'#0¢#65535'"¢":2~'#0
¢#65535'"'~'#0¢#65535'"
```

(The proofs are left as an exercise to the reader.)

Testing for greater-than with 16-bit values is somewhat simpler and is done with the pair of statements:

```
DO .C <- '∀.Ac.B'~'#0¢#65535'
DO .C <- '&"'.A~.C'~'"'∀'∀.C~.C'¢#32768"
      ~"#0¢#65535"'"¢".C~.C"'~#1
```

This sets .C (a dummy variable) to #1 if .A > .B, and #0 otherwise. The expression may be expanded as described above to instead set .C to #1 or #2.

Note also in statement 220 the occurrence of ~"#65535¢#65535". Although these operations select the entire value, they are not extraneous, as they ensure that the forthcoming ∀s will be operating on 32-bit values.

In several virtual computed GO TOs the DO FORGET #1 (statement 15 in the earlier example) has been omitted, since the next transfer of control would be a DO RESUME #1. By making this a DO RESUME #2 instead, the FORGET may be forgotten.

In statement 64, note that .2 is STASHed twice by a single statement. This is perfectly legal.

Lastly, note in statements 243 and 214 respectively, expressions for shifting 16- and 32-bit variables logically one place to the left. Statement 231 demonstrates right-shifting for 32-bit variables.

6.3. Program Listing

```
    1   (1000)  PLEASE IGNORE .4
    2       PLEASE ABSTAIN FROM (1005)
    3   (1009)  DO STASH .1 + .2 + .5 + .6
    4       DO .4 <- #1
    5       DO (1004) NEXT
    6   (1004)  PLEASE FORGET #1
    7       DO .3 <- '¥.1¢.2'~'#0¢#65535'
    8       DO .6 <- '&.1¢.2'~'#0¢#65535'
    9       PLEASE DO .5 <- "¥!6~#32768'¢#1"~#3
   10       DO (1002) NEXT
   11       DO .4 <- #2
   12   (1005)  DO (1006) NEXT
*  13   (1999)  DOUBLE OR SINGLE PRECISION OVERFLOW
   14   (1002)  DO (1001) NEXT
   15   (1006)  PLEASE FORGET #1
   16       DO .5 <- '¥"!6~.6'~#1"¢#1'~#3
   17       DO (1003) NEXT
   18       DO .1 <- .3
   19       DO .2 <- !6¢#0'~'#32767¢#1'
   20       DO (1004) NEXT
   21   (1003)  DO (1001) NEXT
   22       DO REINSTATE (1005)
   23   (1007)  PLEASE RETRIEVE .1 + .2 + .5 + .6
   24       DO REMEMBER .4
   25       PLEASE RESUME #2
   26   (1001)  DO RESUME .5
   27   (1010)  DO STASH .1 + .2 + .4
   28       DO .4 <- .1
   29       DO .1 <- '¥.2¢#65535'~'#0¢#65535'
   30       DO (1020) NEXT
   31       PLEASE DO .2 <- .4
   32       PLEASE DO (1009) NEXT
   33       DO RETRIEVE .1 + .2 + .4
   34       PLEASE RESUME #1
   35   (1020)  DO STASH .2 + .3
   36       DO .2 <- #1
   37       PLEASE DO (1021) NEXT
   38   (1021)  DO FORGET #1
   39       DO .3 <- "¥!1~.2'¢#1"~#3
   40       PLEASE DO .1 <- '¥.1¢.2'~'#0¢#65535'
   41       DO (1022) NEXT
   42       DO .2 <- !2¢#0'~'#32767¢#1'
   43       DO (1021) NEXT
   44   (1023)  PLEASE RESUME .3
   45   (1022)  DO (1023) NEXT
   46       PLEASE RETRIEVE .2 + .3
   47       PLEASE RESUME #2
   48   (1030)  DO ABSTAIN FROM (1033)
   49       PLEASE ABSTAIN FROM (1032)
   50   (1039)  DO STASH :1 + .5
   51       DO (1530) NEXT
   52       DO .3 <- :1~#65535
   53       PLEASE DO .5 <- :1~'#65280¢#65280'
```

```
    DO .5 <- '∀"!5~.5'~#1"¢#1'~#3
    DO (1031) NEXT
(1032) DO (1033) NEXT
    DO (1999) NEXT
(1031) DO (1001) NEXT
(1033) DO .4 <- .5
    DO REINSTATE (1032)
    PLEASE REINSTATE (1033)
    DO RETRIEVE :1 + .5
    PLEASE RESUME #2
(1040) PLEASE STASH .1 + .2 + .2 + :1 + :2 + :3
    DO .2 <- #0
    DO (1520) NEXT
    DO STASH :1
    PLEASE RETRIEVE .2
    DO .1 <- .2
    DO .2 <- #0
    PLEASE DO (1520) NEXT
    DO :2 <- :1
    DO RETRIEVE .1 + .2 + :1
    DO (1550) NEXT
    PLEASE DO .3 <- :3
    DO RETRIEVE :1 + :2 + :3
    DO RESUME #1
(1050) PLEASE STASH :2 + :3 + .5
    DO :2 <- .1
    PLEASE DO (1550) NEXT
    DO .5 <- :3~'#65280¢#65280'
    DO .5 <- '∀"!5~.5'~#1"¢#1'~#3
    DO (1051) NEXT
    DO (1999) NEXT
(1051) DO (1001) NEXT
    DO .2 <- :3
    PLEASE RETRIEVE :2 + :3 + .5
    DO RESUME #2
(1500) PLEASE ABSTAIN FROM (1502)
    PLEASE ABSTAIN FROM (1506)
(1509) PLEASE STASH :1 + .1 + .2 + .3 + .4 + .5 + .6
    DO .1 <- :1~#65535
    PLEASE DO .2 <- :2~#65535
    DO (1009) NEXT
    DO .5 <- .3
    PLEASE DO .6 <- .4
    DO .1 <- :1~'#65280¢#65280'
    DO .2 <- :2~'#65280¢#65280'
    DO (1009) NEXT
    DO .1 <- .3
    PLEASE DO (1503) NEXT
    DO .6 <- .4
    DO .2 <- #1
    DO (1009) NEXT
    DO .1 <- .3
    DO (1501) NEXT
(1504) PLEASE RESUME .6
```

```
108   (1503)  DO (1504) NEXT
109   (1501)  DO .2 <- .5
110       DO .5 <- 'V"'&.6¢.4'~#1"¢#2'~#3
111       DO (1505) NEXT
112   (1506)  DO (1502) NEXT
113       PLEASE DO (1999) NEXT
114   (1505)  DO (1001) NEXT
115   (1502)  DO :4 <- .5
116       DO (1520) NEXT
117       DO :3 <- :1
118       PLEASE RETRIEVE :1 + .1 + .2 + .3 + .4 + .5 + .6
119       DO REINSTATE (1502)
120       DO REINSTATE (1506)
121       PLEASE RESUME #3
122   (1510)  DO STASH :1 + :2 + :4
123       DO :1 <- "'V":2~'#65535¢#0'"¢#65535'~'#0¢#6553
      5'"¢"'V":2~'#0¢#65535'"¢#65535'~'#0¢#65535
      '"
124       DO :2 <- #1
125       DO (1509) NEXT
126       PLEASE RETRIEVE :1
127       DO :2 <- :3
128       PLEASE DO (1509) NEXT
129       DO RETRIEVE :2 + :4
130       PLEASE RESUME #1
131   (1520)  PLEASE STASH .3 + .4
132       DO .3 <- .1~#43690
133       DO (1525) NEXT
134       PLEASE DO .4 <- 'V.3¢".2~#43690"'~'#0¢#65535'
135       DO .3 <- .1~#21845
136       PLEASE DO (1525) NEXT
137       DO :1 <- .4¢"'V.3¢".2~#21845"'~'#0¢#65535'"
138       PLEASE RETRIEVE .3 + .4
139       DO RESUME #1
140   (1525)  DO .3 <- '"'"'"!3¢#0'~'#32767¢#1'"¢#0'~'#32767
      ¢#1'"¢#0'~'#16383¢#3'"¢#0'~'#4095¢#15'
141       PLEASE RESUME #1
142   (1530)  DO STASH :2 + :3 + .3 + .5
143       DO :1 <- #0
144       DO :2 <- .2
145       DO .3 <- #1
146       DO (1535) NEXT
147   (1535)  PLEASE FORGET #1
148       DO .5 <- "V!1~.3'¢#1"~#3
149       DO (1531) NEXT
150       DO (1500) NEXT
151       DO :1 <- :3
152       PLEASE DO (1533) NEXT
153   (1531)  PLEASE DO (1001) NEXT
154   (1533)  DO FORGET #1
155       DO .3 <- !3¢#0'~'#32767¢#1'
156       DO :2 <- ":2~'#0¢#65535'"¢"'":2~'#32767¢#0'"¢#
      0'~'#32767¢#1'"
157       PLEASE DO .5 <- "V!3~.3'¢#1"~#3
```

```
158       DO (1532) NEXT
159       DO (1535) NEXT
160   (1532) DO (1001) NEXT
161       PLEASE RETRIEVE :2 + :3 + .3 + .5
162       DO RESUME #2
163   (1540) PLEASE ABSTAIN FROM (1541)
164       DO ABSTAIN FROM (1542)
165   (1549) PLEASE STASH :1 + :2 + :4 + :5 + .1 + .2 + .5
166       DO .1 <- :1~#65535
167       PLEASE DO .2 <- :2~'#65280¢#65280'
168       DO .5 <- :1~'#65280¢#65280'
169       DO (1530) NEXT
170       DO :3 <- :1
171       DO .2 <- :2~#65535
172       PLEASE DO (1530) NEXT
173       DO :5 <- :1
174       DO .1 <- .5
175       DO (1530) NEXT
176       DO :4 <- :1
177       PLEASE DO :1 <- ":3~'#65280¢#65280'"¢":5~'#652
      80¢#65280'"
178       DO .5 <- ':1~:1'~#1
179       DO .2 <- :2~'#65280¢#65280'
180       DO (1530) NEXT
181       PLEASE DO .5 <- '"':1~:1'~#1"¢.5'~#3
182       DO .1 <- :3~#65535
183       DO .2 <- #0
184       DO (1520) NEXT
185       PLEASE DO :2 <- :1
186       PLEASE DO .1 <- :4~#65535
187       DO (1520) NEXT
188       DO (1509) NEXT
189       DO .5 <- !5¢":4~#3"'~#15
190       DO :1 <- :3
191       DO :2 <- :5
192       DO (1509) NEXT
193       PLEASE DO .5 <- !5¢":4~#3"'~#63
194       DO .5 <- '∀"!5~.5'~#1"¢#1'~#3
195       PLEASE RETRIEVE :4
196   (1541) DO :4 <- .5
197       DO (1543) NEXT
198   (1542) DO (1544) NEXT
199       PLEASE DO (1999) NEXT
200   (1543) DO (1001) NEXT
201   (1544) DO REINSTATE (1541)
202       PLEASE REINSTATE (1542)
203       PLEASE RETRIEVE :1 + :2 + :5 + .1 + .2 + .5
204       DO RESUME #2
205   (1550) DO STASH :1 + :4 + :5 + .5
206       DO :3 <- #0
207       DO .5 <- '∀"':2~:2'~#1"¢#1'~#3
208       PLEASE DO (1551) NEXT
209       DO :4 <- #1
210       PLEASE DO (1553) NEXT
```

```
    211   (1553)  DO FORGET #1
    212           DO .5 <- 'V":2~'#32768¢#0'"¢#2'~#3
    213           DO (1552) NEXT
    214           DO :2 <- ":2~'#0¢#65535'"¢"':2~'#32767¢#0'"¢#
                  0'~'#32767¢#1'"
    215           PLEASE DO :4 <- ":4~'#0¢#65535'"¢"':4~'#32767
                  ¢#0'"¢#0'~'#32767¢#1'"
    216           DO (1553) NEXT
    217   (1552)  DO (1001) NEXT
    218   (1556)  PLEASE FORGET #1
    219           DO :5 <- "'V":1~'#65535¢#0'"¢":2~'#65535¢#0'"'
                  ~'#0¢#65535'"¢"'V":1~'#0¢#65535'"¢":2~'#0¢
                  #65535'"'~'#0¢#65535'"
    220           DO .5 <- 'V"'&"':2~:5'~'"'V"'V":5~:5"~"#65535~
                  #65535"'~'#65535¢#0'"¢#32768'~'#0¢#65535'"
                  ¢"'V":5~:5"~"#65535¢#65535"'~'#0¢#65535'"'
                  "¢"':5~:5'~#1"'~#1"¢#2'~#3
    221           DO (1554) NEXT
    222           DO :5 <- :3
    223           DO (1510) NEXT
    224           PLEASE DO :1 <- :3
    225           DO :3 <- "'V":4~'#65535¢#0'"¢":5~'#65535¢#0'"'
                  ~'#0¢#65535'"¢"'V":4~'#0¢#65535'"¢":5~'#0¢
                  #65535'"'~'#0¢#65535'"
    226           DO (1555) NEXT
    227   (1554)  PLEASE DO (1001) NEXT
    228   (1555)  DO FORGET #1
    229           DO .5 <- "V':4~#1'¢#2"~#3
    230           DO (1551) NEXT
    231           DO :2 <- ":2~'#0¢#65534'"¢":2~'#65535¢#0'"
    232           DO :4 <- ":4~'#0¢#65534'"¢":4~'#65535¢#0'"
    233           PLEASE DO (1556) NEXT
    234   (1551)  DO (1001) NEXT
    235           PLEASE RETRIEVE :1 + :4 + :5 + .5
    236           PLEASE RESUME #2
    237   (1900)  DO STASH .2 + .3 + .5
    238           DO .1 <- #0
    239           DO .2 <- #1
    240           PLEASE DO (1901) NEXT
    241   (1901)  DO FORGET #1
    242           DO %50 .1 <- 'V.1¢.2'~'#0¢#65535'
    243           DO .2 <- !2¢#0'~'#32767¢#1'
    244           PLEASE DO .5 <- "V!2~.2'¢#1"~#3
    245           DO (1902) NEXT
    246           DO (1901) NEXT
    247   (1902)  DO (1001) NEXT
    248           DO RETRIEVE .2 + .3 + .5
    249           PLEASE RESUME #2
    250   (1910)  PLEASE STASH .1 + .3 + .5 + :1 + :2 + :3
    251           DO .3 <- #65524
    252           DO :1 <- #6
    253           DO (1911) NEXT
*   254           PLEASE NOTE THAT YOU CAN'T GET THERE FROM HERE
    255   (1912)  DO (1001) NEXT
```

```
256   (1911)  DO FORGET #1
257       PLEASE DO (1900) NEXT
258       DO :2 <- .1
259       DO (1500) NEXT
260       PLEASE DO :1 <- :3
261       DO .1 <- .3
262       DO (1020) NEXT
263       PLEASE DO .3 <- .1
264       DO .5 <- 'V"!3~.3'~#1"¢#2'~#3
265       DO (1912) NEXT
266       DO .1 <- #12
267       PLEASE DO (1050) NEXT
268       DO RETRIEVE .1
269       DO (1530) NEXT
270       DO :2 <- #32768
271       DO (1500) NEXT
272       PLEASE DO .2 <- :3~'#65280¢#65280'
273       PLEASE RETRIEVE .3 + .5 + :1 + :2 + :3
274       DO RESUME #1
```

6.4. Programming Suggestions

For the novice INTERCAL programmer, we provide here a list of suggested INTERCAL programming projects:

Write an integer exponentiation subroutine. :1 <- .1 raised to the .2 power.

Write a double-precision sorting subroutine. Given 32-bit array ;1 of size :1, sort the contents into numerically increasing order, leaving the results in ;1.

Generate a table of prime numbers.

Put together a floating-point library, using 32-bit variables to represent floating-point numbers (let the upper half be the mantissa and the lower half be the characteristic). The library should be capable of performing floating-point addition, subtraction, multiplication, and division, as well as the natural logarithm function.

Program a Fast Fourier Transform (FFT). This project would probably entail the writing of the floating-point library as well as sine and cosine functions.

Calculate, to :1 places, the value of pi.

7. ERROR MESSAGES

Due to INTERCAL's implementation of comment lines (see section 4.5), most error messages are produced during execution instead of during compilation. All errors except those not causing immediate termination of program execution are treated as fatal.

7.1. Format

All error messages appear in the following form:

```
ICLnnnI (error message)
        ON THE WAY TO STATEMENT nnnn
        CORRECT SOURCE AND RESUBMIT
```

The message varies depending upon the error involved. For undecodable statements the message is the statement itself. The second line tells which statement would have been executed next had the error not occurred. Note that if the error is due to 80 attempted levels of NEXTing, the statement which would have been executed next need not be anywhere near the statement causing the error.

7.2. Messages

Brief descriptions of the different error types are listed below according to message number.

000 An undecodable statement has been encountered in the course of execution. Note that keypunching errors can be slightly disastrous, since if 'FORGET' were misspelled F-O-R-G-E-R, the results would probably not be those desired. Extreme misspellings may have even more surprising consequences. For example, misspelling 'FORGET' R-E-S-U-M-E could have drastic results.

017 An expression contains a syntax error.

079 Improper use has been made of statement identifiers.

099 Improper use has been made of statement identifiers.

123 Program has attempted 80 levels of NEXTing.

129 Program has attempted to transfer to a non-existent line label.

139 An ABSTAIN or REINSTATE statement references a non-existent line label.

182 A line label has been multiply defined.

197 An invalid line label has been encountered.

200 An expression involves an unidentified variable.

240 An attempt has been made to give an array a dimension of zero.

241 Invalid dimensioning information was supplied in defining or using an array.

275 A 32-bit value has been assigned to a 16-bit variable.

436 A retrieval has been attempted for an unSTASHed value.

533 A WRITE IN statement or interleave (¢) operation has produced a value requiring over 32 bits to represent.

562 Insufficient data.

579 Input data is invalid.

621 The expression of a RESUME statement evaluated to #0.

632 Program execution was terminated via a RESUME statement instead of GIVE UP.

633 Execution has passed beyond the last statement of the program.

774 A compiler error has occurred (see section 8.1).

778 An unexplainable compiler error has occurred (see J. Lyon or D. Woods).

8. JCL

The information contained in the following section applies only to the Princeton compiler, run under OS/360.

8.1. The Princeton Compiler

The Princeton compiler, written in SPITBOL (a variant of SNOBOL), performs the compilation in two stages. First the INTERCAL source is converted into SPITBOL source, then the latter is compiled and executed.

It should be noted that the Princeton compiler fails to properly interpret certain multiply-subscripted expressions, such as:

```
",1SUB",2SUB#1"#2"
```

This is not a "bug". Being documented, it is merely a "restriction". Such cases may be resolved by alternating sparks and ears in various levels of expression nesting:

```
",1SUB',2SUB#1'#2"
```

which is advisable in any case, since INTERCAL expressions are unreadable enough as is.

Since there is currently no catalogued procedure for invoking the compiler, the user must include the in-line procedure shown on the following page in his job before the compilation step. Copies of this in-line procedure may be obtained at any keypunch if the proper keys are struck.

The compiler is then executed in the usual manner:

```
// EXEC INTERCAL[,PARM='parameters']
//COMPILE.SYSIN DD *
{INTERCAL source deck}
/*
//EXECUTE.SYSWRITE DD *
{input data}
/*
```

The various parameters are described following the in-line procedure. At most one parameter from each set may apply to a given compilation; if more than one are specified, the results are undefined, and may vary depending upon the particular set of options. The default parameters are underlined.

```
//INTERCAL PROC
//COMPILE EXEC PGM=INTERCAL
//STEPLIB DD DSN=U.INTERCAL.LIBRARY,DISP=SHR
//          DD DSN=SYS1.FORTLIB,DISP=SHR
//SYSPRINT DD SYSOUT=A,DCB=(BLKSIZE=992,LRECL=137,RECFM=VBA)
//SYSPUNCH DD DUMMY
//SCRATCH DD DSN=&COMPSET,UNIT=SYSDA,SPACE=(CYL,(3,1)),DISP=(,PASS)
//EXECUTE EXEC PGM=EXECUTE,COND=(4,LT)  [2]
//SOURCES DD DSN=U.INTERCAL.SOURCES,DISP=SHR
//STEPLIB DD DSN=U.INTERCAL.LIBRARY,DISP=SHR
//          DD DSN=SYS5.SPITLIB,DISP=SHR
//          DD DSN=SYS1.FORTLIB,DISP=SHR
//SYSIN DD DSN=&COMPSET,DISP=(OLD,DELETE)
//SYSOBJ DD SYSOUT=B,DCB=(BLKSIZE=80,LRECL=80,RECFM=F)
//SYSPRINT DD SYSOUT=A,DCB=(BLKSIZE=992,LRECL=137,RECFM=VBA)
//SYSPUNCH DD DUMMY
// PEND
```

Figure 3. Inline procedure for using INTERCAL.

OPT
NOOPT

In the default mode, the compiler will print a list of all options in effect, including the defaults for unspecified parameter groups and the effective option for those sets where one was specified. If NOOPT is requested, it causes the default mode to be assumed.

OPTSUB
NOOPTSUB
NOSUB

Unless 'NOOPTSUB' is requested, the System Library is optimized, resulting in much more rapid NOSUB processing of function calls. Specifying NOOPTSUB causes the non-optimized INTERCAL code shown in section 6.3 to be used, whereas NOSUB requests that the System Library be omitted altogether.

IAMBIC
PROSE

The IAMBIC parameter permits the programmer to use poetic license and thus write in verse. If the reader does not believe it possible to write verse in INTERCAL, he should send the authors a stamped, self-addressed envelope, along with any INTERCAL program, and they will provide one which is verse.

FORMAT
NOFORMAT

In FORMAT mode, each statement printed is put on a separate line (or lines). In NOFORMAT mode, the free-format source is printed exactly as input. In this latter case, statement numbers are provided only for the first statement on a card, and they may be only approximate. Also, unrecognizable statements are not flagged.

SEQ
NOSEQ

---

[2] Pending acquisition of SPITBOL release 3.0, the SOURCES DD card must be replaced by the five cards:

```
//NOOPTPFX DD DSN=U.INTERCAL.SOURCES(NOOPTPFX),DISP=SHR
//NOOPTSUB DD DSN=U.INTERCAL.SOURCES(NOOPTSUB),DISP=SHR
//OPTPFX DD DSN=U.INTERCAL.SOURCES(OPTPFX),DISP=SHR
//OPTSUB DD DSN=U.INTERCAL.SOURCES(OPTSUB),DISP=SHR
//PRELIM DD DSN=U.INTERCAL.SOURCES(PRELIM),DISP=SHR
```

If the source deck has sequence numbers in columns 73 through 80, specifying 'SEQ' will cause them to be ignored.

SOURCE
NOSOURCE

If NOSOURCE is selected, all source listing is suppressed.

LIST
NOLIST

If LIST is specified, the compiler will provide a list of statement numbers catalogued according to type of statement. The compiler uses this table to perform abstentions by gerund.

WIDTH=nn

This sets the width (in number of characters) of the output line for FORMAT mode output. The default is **132.**

CODE
NOCODE

Include 'CODE' in the parameter list to obtain a listing of the SPITBOL code produced for each INTERCAL statement.

LINES=nn

This determines the number of lines per page, during both compilation and execution. The default is **60.**

DECK
NODECK

Selecting 'DECK' will cause the compiler to punch out a SPITBOL object deck which may then be run without reinvoking the INTERCAL (or SPITBOL) compiler.

KIDDING
NOKIDDING

Select NOKIDDING to eliminate the snide remarks which ordinarily accompany INTERCAL error messages.

GO
NOGO

Specifying 'NOGO' will cause the program to be compiled but not executed. EXECUTE/NOEXECUTE may be substituted for GO/NOGO, but this will result in an error, and GO will be assumed.

BUG
NOBUG

Under the default, there is a fixed probability of a fatal compiler bug being worked at random into the program being compiled. Encountering this bug during execution results in error message 774 (see section 7.2). This probability is reduced to zero under 'NOBUG'. This does not affect the probability (presumably negligible) of error message 778.

8.2. Other INTERCAL Compilers

There are no other INTERCAL compilers. [3]

---

[3] Note that this document pre-dates Eric Raymond's C-INTERCAL compiler, and an Atari implementation mentioned on the final page of this document.

The Official INTERCAL Character Set

Tabulated on page 28 are all the characters used in INTERCAL, excepting letters and digits, along with their names and interpretations. Also included are several characters not used in INTERCAL, which are presented for completeness and to allow for future expansion.

[4] Since all other reference manuals have Appendices, it was decided that the INTERCAL manual should contain some other type of removable organ.

[5] This footnote intentionally unreferenced.

box; c c c l l l. Character NameUse (if any) =

: two-spot identify 32-bit variable , tail identify 16-bit array ; hybrid identify

32-bit array # mesh identify constant = half-mesh ‘ backspark ! wow equivalent to spark-spot ? what *unary exlusive OR (ASCII)* " rabbit-ears grouper ‘. rabbit equivalent to ears-spot | spike % double-oh-seven percentage qualifier – worm used with angles < angle used with worms > right angle (wax precedes line label) wane follows line label [U turn] U turn back { embrace } bracelet * splat flags invalid statements & ampersand[5] unary logical AND V V (or book) unary logical OR ∀ bookworm (or universal qualifier) unary exclusive OR $ big money *binary mingle (ASCII)* ¢ change binary mingle ~ sqiggle binary select _ flat worm ¯ overline indicates "times 1000" + intersection separates list items / slat \ backslat @ whirlpool ¬ hookworm ^ shark (or simply sharkfin) ▩ blotch

Table 2 (top view). INTERCAL character set.

[5] Got any better ideas?

NOTES ON THE ATARI IMPLEMENTATION

The Atari implementation of INTERCAL differs from the original Princeton version primarily in the use of ASCII rather than EBCDIC. Since there is no "change" sign (¢) in ASCII, we have substituted the "big money" ($) as the mingle operator. We feel that this correctly represents the increasing cost of software in relation to hardware. (Consider that in 1970 one could get RUNOFF for free, to run on a $20K machine, whereas today a not quite as powerful formatter costs $99 and runs on a $75 machine.) We also feel that there should be no defensible contention that INTERCAL has any sense. Also, since overpunches are difficult to read on the average VDT, the exclusive-or operator may be written ?. This correctly expresses the average person's reaction on first encountering exclusive-or, especially on a PDP-11. Note that in both of these cases, the over-punched symbol may also be used if one is masochistic, or concerned with portability to the Princeton compiler. The correct over-punch for "change" is "c<backspace>/" and the correct over-punch for ∀ is "V<backspace>-". These codes will be properly printed if you have a proper printer, and the corresponding EBCDIC code will be produced by the /IBM option on the LIST command.